isekai Technical Documentation

Guillaume Drevon, Nemanja Borić

July 2019 - version 0.9

Contents

1 Introduction **3**

2 Usage **3**

2.1 Input . . . 3

2.2 R1CS . . . 3

2.3 Proof . . . 4

2.4 Verification . . . 4

3 Architecture **4**

3.1 Frontend . . . 5

3.1.1 AST reduction . . . 5

3.1.2 Internal Representation . . . 6

3.1.3 Collapser . . . 7

3.1.4 Frontend Parser . . . 8

4 Backend **12**

4.1 Arithmetic Circuit . . . 12

4.2 Bus . . . 13

4.3 Trace, FieldOp and Wire . . . 14

4.4 Board . . . 15

4.4.1 Bit Width . . . 15

4.5 Inputs . . . 16

4.6 Wire Ordering . . . 17

4.7 Bus request and factory . . . 17

4.8 Backend Flow . . . 18

4.9 R1CS reduction . . . 19

4.10 Prover . . . 20

4.11 Next steps . . . 20

4.11.1 Programming languages . . . 20

4.11.2 High level language features . . . 20

4.11.3 Zero-Knowledge Libraries . . . 21

4.11.4 summary . . . 21

1 Introduction

isekai is a framework for zero-knowledge proofs applications. It aims at providing easy-to-use interfaces over most zero-knowledge proofs implementations and to plug them with programs written by regular software engineers (i.e not specialist in this field). For now isekai only works with program written in C and can only generate zero-knowledge succinct non-interactive argument of knowledge (*zk-SNARKs*). This technical documentation explains how to use isekai, how it is working under the hood and how we want to it to evolve.

2 Usage

isekai is used to generate or verify a **proof** of the execution of a computer program (C function).

2.1 Input

The computer program is a C function having one of the following signature

```
void outsource(struct Input *input, struct NzikInput * nzik, struct Output *output);

void outsource(struct Input *input, struct Output *output);

void outsource(struct NzikInput * nzik, struct Output *output);
```

Input and Output are public parameters and NzikInput are the private parameters (zero-knowledge). Inputs and NzikInputs can be provided in an additional file, by putting each value one per line. This input file must have the same name as the C program file, with an additional '.in' extension. For instance, if the function is implemented in `my_prog.c`, the inputs must be provided in `my-prog.c.in`.

2.2 R1CS

From the C source code, you can generate corresponding R1CS representation in json format [Dre19]. This is done with isekai using the following command line:

```
isekai --r1cs=my_prog.j1 my_prog.c
```

In this example, the outsource function is implemented in the `my_prog.c` file, which is given as input to isekai. The result of the command (output) is `my_prog.j1`. Of course you can name it whatever you want by providing the desired name in the –r1cs argument. isekai also generates the assignments in the file `my_prog.j1.in`. It adds '.in' to the filename provided in the r1cs option to get a filename for the assignments. Note that existing files are overwritten by isekai.

isekai automatically uses the inputs provided in `my_prog.c.in` if this file exists. If not, isekai assumes all the inputs are 0.

Since we currently support only a limited set of C features, the R1CS reduction will fail for most of the programs. For instance, arrays and float are not supported.

2.3 Proof

Once you have the R1CS and assignments files (in our example they are the .j1 and .j1.in files), you can use them to create a trusted setup and a (zk-snark) proof:

```
isekai --snark=my_prog my_prog.j1
```

This command generates two files `my_prog.s` and `my_prog.p`. They use the prefix provided in the –snark option and add .s/.p to it. `my_prog.s` is the trusted setup and `my_prog.p` is the proof of execution. Indeed, the function has been executed during the R1CS reduction for generating the assignments. Note that in real life, you should not generate a proof and the trusted setup at the same time!

2.4 Verification

A verifier can verify the proof with the following command:

```
isekai --verify=my_prog my_prog.j1.in
```

Here you must have `my_prog.s`, `my_prog.p` and `my_prog.j1.in` as input to the command. A verifier should not know the private inputs (NzikInput) so you should remove the 'witnesses' part from the file `my_prog.j1.in`.

3 Architecture

Conceptually and code-wise, isekai is split into four parts. It has been designed in order to be able to facilitate the support of additional programming language for the input source code, the integration of several zero-knowledge proof libraries and the implementation of common high-level programming language features.

- The "**frontend**" part that parses C code and transforms it into the intermediate form (expression graph).
- The "**backend**" part which takes the expression graph produced by the frontend and produces either arithmetic or boolean circuit.
- The "**reducer**" which takes an arithmetic circuit and produces R1CS.
- The "**prover**" part that takes R1CS and generate a proof of execution.

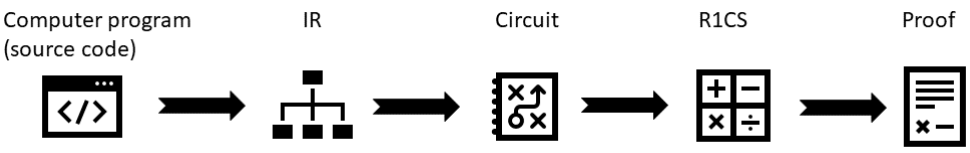

Figure 1: Highlevel isekai workflow

The figure 1 shows the work-flow between the different parts. The input and output of each parts are very well defined and follows a waterfall pattern from the first part to the last, so that they are easy to modify independently. As a result, one can add additional programming language support in the first part, additional zk proof scheme in the last one and additional programming features in the third one (and the second one to a lesser extent).

isekai is implemented using Crystal programming language [Man19]. As stated in [IB19]; *Crystal combines the elegant coding of dynamic languages like Ruby or Python with the safety and blazing performance of a natively compiled language like Go or Rust. Ruby - a very similar language - is sometimes said to be a programmer's best friend. Crystal is a language for both humans and computers.*

3.1 Frontend

The purpose of the frontend is to take the input source code and transform it into isekai internal representation (IR). This part involves two steps, the first one being the generation of the abstract syntax tree (AST).

3.1.1 AST reduction

The first step of the frontend is to transforms a C program into an Abstract Syntax Tree. This operation is typically done inside a compiler, using Shunting-yard algorithm, so it should be common for any compiled language. In isekai, we support only the C language for now and the reduction into AST is done with the `libclang` library, which is a C wrapper around clang compiler. In the code, this is done at the beginning of the `Cparser.parse()` function (in `parser.cr`), with the call to `ClangUtils.parse_file_to_ast_tree(...)` which returns a Clang translation unit. More information can be found at Clang documentation [Tea19].

Recall that the input C program must be of the following form:

- `void outsource(struct Input *input, struct Output *output);`

Or, with zero-knowledge inputs:

- `void outsource(struct Input *input, struct NzikInput * nzik, struct Output *output);`

The `outsource` function is the main entry point to the program, it is the function we want to prove the execution and it must be called `outsource`.

As `libclang` is a C library, it's trivially accessible from crystal language, and for that purpose `clang.cr` bindings are used which provide crystal API around the library. Unfortunately, not all features of `libclang` are available in the upstream version, so several patches are applied on it for several small functionalities (e.g. the way to get the initial, conditional and increment part of the for statement). To ease the debugging and developing, isekai provides a docker image with all dependencies resolved.

Some parts of frontend are currently tightly connected with `libclang`'s interface and parsing C code. This is true for all the methods that transform the declaration, control structures and statements from C to internal representation. `decode_type` would be a good example, where we translate the `libclang`'s interface. But `transform_if`, `transform_loop`, `nroll_dynamic_loop`, `unroll_static_loop`, `Collapser`, `evaluate()` and the entire middle layer should not needed to be modified for any language (of course, if the language provide these statements). And this is where the most complexity of the frontend lies - declaration of the variables, types, decoding the tokens in the control structures are not complicated operations. If the frontend is needed for any language, making a visitor method (akin to clang's `visit_children`) should be a preferred approach. That will allow one to simply keep most of the functionality. Alternatively, as long as `decode_expression` can be used with another language, all transformation functions should work.

If the new language has large difference in structure, the most important thing should be not changing the middle layer. As long as the middle layer is the same (i.e as long as the frontend produces the DFGExpr - see the next section to describe the connection between the input and the output), at least the same backend can be used. And again, most of the transformation functions can be reused.

3.1.2 Internal Representation

The AST is finally transformed into the isekai internal representation with the call to `create_expression()`. The result is of type State, containing the tree (DFGExpr) and the symbol table (SymbolTable).

The basic building block of the intermediate representation is an expression node, called `DFGExpr` in the code. It is defined in dfg.cr and it's basically an abstract class. It inherits from `SymbolTableValue` which is an abstract class defining an entry in the symbol table (types and expresion). It provides the symbol size: sizeof the returns number of bytes for the type (-1 for abstract entry). Every operation or entity can be modeled by a single instance of the node. Example node types would be Constant, Conditional, Multiply, etc. The final expression is made by composing the nodes.

Symbol Table: In general, a symbol table is a dictionary mapping symbol to their definition and can tell whether it is a variable or a function. Several patterns for constructing a symbol table exist, depending on how the scope is managed in the language.

SymbolTableValue: is an abstract class defining an entry in the symbol table (types and expresion). It provides the symbol size: sizeof the returns number of bytes for the type (-1 for abstract entry). A symbol table entry holds two classes of keys; Symbol (a variable) and StorageKey (variable instance). Key ←Symbol, StorageKey

Consider the following pseudo-C code:

```
output->x = (input->a / 10) == (input->b + 20);
```

Generated tree for the output symbol would look like:

```
#<Isekai::CmpEQ:0x7fa3e1cbc420
 @left=
  #<Isekai::Divide:0x7fa3d7b22d00
   @crystalop=#<Isekai::OperatorDiv:0x7fa3e1d5d0f0>,
   @identity=1,
   @left=
#<Isekai::Input:0x7fa3e1cc9000
  @storage_key=
   #<Isekai::StorageKey:0x7fa3e1cc9020
    @idx=0,
    @storage=#<Isekai::Storage:0x7fa3e1cc9760 @label="input", @size=2>>>,
   @op="/",
   @right=#<Isekai::Constant:0x7fa3e1d5d110 @value=10>>,
 @op="==",
 @right=
  #<Isekai::Add:0x7fa3d7b263c0
   @crystalop=#<Isekai::OperatorAdd:0x7fa3e1d5d080>,
   @identity=0,
   @left=
#<Isekai::Input:0x7fa3d7b21fe0
  @storage_key=
   #<Isekai::StorageKey:0x7fa3e1cc9de0
    @idx=1,
    @storage=#<Isekai::Storage:0x7fa3e1cc9760 @label="input", @size=2>>>,
   @op="+",
   @right=#<Isekai::Constant:0x7fa3e1d5d0a0 @value=20>>>
```

Output snippet is a tree built from several subtrees. Highlighted parts are the left and the right side of the CmpEQ node. Left side encodes `input->a / 10` and the right side `input->b + 20`.

3.1.3 Collapser

Collapser is the tool that's used both in the backend and in the frontend part of the compiler. It's purpose is to collapse down the expressions to the minimal form and/or constant form. The Collapser's main functionality is found in `Collapser.collapse_tree function`. The main flow is very simple and it's done in a several steps:

1. Get the dependencies of the given expression (for example, Add expression would have left and right operands as a dependencies)
2. Lookup all the dependencies in the list of already calculated values and grab the new ones.
3. For each new dependency recursively do the same steps

4. One all dependencies are resolved, evaluate the expression calling `DFGExpr.valuate` which is overridden for every type of the expression (e.g. evaluation of Add is different to evaluation of Division).

3.1.4 Frontend Parser

Cparser is defined in parser.cr and has the following fields:

- `@root_func_cursor`: pointer to the 'outsource' function (entry point)
- `@output`: array of output parameters (as tuple of StorageKey, DFGExpr)
- `@inputs`: array of input parameters ('instance', as DFGExpr)
- `@nizk_inputs`: array of private parameters ('witness', as DFGExpr)

The core of the frontend is the `CParser` class. Responsibilities of the class include setting up the `libclang` handle, parsing the C file and visiting the entire AST tree of the C file and transforming it to intermediate format. Since the parsing is a statefull procedure, the entire state is stored in an instance of `SymbolTable`. The parser holds the instance of the global symbol table and they are chained - symbol table is a parent of all symbol tables in the child scopes (using C's definition of a scope). So, global symbol table is a parent of every symbol table and the parser instance can lookup any symbol defined (symbol table provides lookup method which behaves according to C's scoping rules - if a symbol is not found in the current scope, we'll lookup in the parent scopes for it until we exhaust all scopes).

Creation of the global symbol table is done inside function `create_global_symtab`. The function creates the instance of the symbol table and then does the most common operation in the frontend - visiting the AST tree. Example of the libclang's API on the next page shows the general idea: on the instance of the AST cursor we call `visit_children method`, passing a delegate which accepts the single argument (new cursor - pointing to the first child of the starting cursor) and which should return one of the following value in order to indicate how to proceed:

```
ChildVisitResult::{Break, Continue, Recurse}
```

Should we break the iteration, should we continue with the sibling, or should we recursively visit the cursor's children? Here we're going just to look into the AST nodes in the main scope and we'll declare variable if we find any and we'll save the cursor of our "main" function `outsource`

The code for visiting children of a given AST node is given below:

```
root.visit_children do |cursor|
    case cursor.kind
    when .var_decl?
       symtab = declare_variable(cursor, symtab)
    when .function_decl?
```

```
      if @root_func_cursor.is_a? Nil && cursor.spelling == "outsource"
       Log.log.debug "Found root function declaration"
       @root_func_cursor = cursor
      end
    when .macro_definition?, .inclusion_directive?, .struct_decl?
      Log.log.debug "ignoring #{cursor.kind} for now: #{cursor}"
    else
      pp cursor
      raise "Found unexpected element in AST, was expecting a declaration"
    next Clang::ChildVisitResult::Break
    end
    next Clang::ChildVisitResult::Continue
end
```

Once our iterator finds the variable declaration (when it visits the AST node with kind `var_decl`), we pass the node to `declare_varible` method which does two things: decodes the type of the variable and maps it to one of the internal types (refer to `src/frontend/types.cr` for the list of types) and it creates the entry in the symbol table by calling its method `declare`. Let's look briefly how we generate a struct type as one of the most complicated examples:

```
    when .struct_decl?
      # struct declaration - create a new struct type
      # and add it to the symbol table
      struct_fields = Array(StructField).new

      ClangUtils.getStructFields(cursor).each do |field_cursor|
       # recursively resolve this field
       Log.log.debug "Decoding struct cursor #{field_cursor}"
       type_state = decode_type(field_cursor.type, symtab, true)
       symtab = type_state.@symtab
       struct_fields << StructField.new(type_state.@type, field_cursor.spelling)
      end

      # against the code that uses structs.
      struct_type = StructType.new(cursor.spelling, struct_fields)
      symtab.declare(Symbol.new(cursor.spelling), struct_type)
      result = TypeState.create(struct_type, symtab)
```

The idea is to collect all struct fields into an array, which we do dynamically calling `decode_type` and once everything is resolved, we do the following:

1. Create a new instance of StructType
2. Create a new instance of Symbol with the name same as the struct type
3. Use symtable.declare to make the parser aware of this type (we declare a type) available under name we created in the second step (which is the same as the struct name).

4. We return the pointer to this type back to the `declare_variable` which will use it to create a new variable of this type.

Once we setup the global symbol table we're ready to transform the body of the root function into the intermediate language graph we showed earlier. The root function has the following signature and it always have the same name:

```
void outsource(struct Input *input, struct Output *output)
```

The ultimate parser's goal is to transform the statements in the outsource function into an expression which will describe output's dependency on the input (refer to the "Intermediate representation" section shown earlier for a simple example of such transformation). In order to transform list of the statements in the single expression, parser's parse method calls `root_funccall` which sets up the symbols for the input and output arguments and finally call `transform_compound`. The sole purpose of this function is to split the compound statement into the list of statements (again, using `visit_children`) pattern and to call `transform_statement` on every statement which will in turn call the appropriate method depending on the type of the statement (e.g. when we hit the if statement, `transform_statement` will call `transform_if`).

Every `transform_*` will eventually call `decode_expression` method. This method maps non-control statements (such as binary expression, unary expression, struct field access, etc) into the internal representation. The next example shows how :

```
    when .unary_operator?
    # Get the child of the unary operator - the expression on which
# the operator is applied
        child = ClangUtils.getFirstChild(expression)
        raise "Unary expression incomplete" unless !child.is_a? Nil

        case expression.operator_str
         when "-"
         state = decode_expression_value(child, symtab)
         return State.new(dfg(Negate, state.expr), state.symtab)
```

In this example, we have detected that we deal with the unary operator (by inspecting the cursor's kind). Next, we'll get the operand which is a first child of the operator, and we'll create a new intermediate graph node based on the operator. In the given example, if negate operator is found, we'll decode an operand's expression (as the entire expression can be -(x-1), in which case we need to recursively decode (x-1) and then apply negate operator on the expression). Once the operand is fully exposed, we'll create a Negate node in the graph (using the helper dfg function).

The most interesting part of the frontend is collapsing/unrolling functionality. The idea is to try to minimize the intermediate graph representation to the minimal. A good example would be an if-statement. In case where isekai can prove during the evaluation that only one of the branches is always taken, the other branch would be completely removed from the code. As an example, let's look into transform_if method.

First, we'll use `ClangUtils.getIfStatementExprs` to get if statement components - an array of two or three elements - a condition, a then branch and an optional else branch.

After that, to see if we can reduce if statement to just one branch. We'll do that by evaluating the condition, which is done in two steps:

1. Convert the condition expression into the internal state. This is done in an usual way, calling `decode_expression_value`
2. Once we have the expression, we'll use the aforementioned `Collapser` to try to evaluate the expression to a constant.
3. If we can evaluate value to the constant expression, we'll just return a single branch, depending on the value.

```
if_parts = ClangUtils.getIfStatementExprs(if_statement)
cond_state = decode_expression_value(if_parts[0], symtab)

begin
  cond_val = evaluate(cond_state.@expr)
    if cond_val != 0
        return transform_statement(if_parts[1], symtab)
    elsif if_parts.size == 3
        return transform_statement(if_parts[2], symtab)
    else
        return symtab #noop
end
```

Now, if any of the evaluating steps fails, collapser will throw a `NonconstantExpression` exception. This would make parser to fully expand if statement, and that's done in the following steps:

1. Create a scope, one for each branch
2. Apply `transform_statement` to each of these and get all identifiers/expressions that are present in the current scope, but are changed in any of the scopes (since nothing else can have any effect on the output).
3. We'll then wrap every symbol in the Conditional node (a member of the intermediate language) and we'll fill the new expression found in both branches to all branches of the Conditional node. This way, at the output, we'll get the new value of the symbol, depending on the conditional value.

Similar approach is done when handling loops. We'll try to evaluate the conditional and to reduce it to a constant value that doesn't depend on any value within the loop. If that's possible, then it's possible just to transform the loop body n times where n is the constant to which the conditional part has evaluated into. If that's not possible, we'll fall back to the dynamic transformation, which we'll explain later.

The entry point for loop's transformation is `transform_loop` method which tries to evaluate the condition early (i.e. before entering the loop's body). If that's possible, we're moving to where we'll perform `unroll_static_loop` static loop evaluation or to `unroll_dynamic_loop` where we'll do a dynamic unrolling.

The first method is very simple - it will perform several steps in the loop:

1. Evaluate the condition's value. If it's false, break the loop
2. Otherwise, evaluate the loop's body and the increment expression.

In case of the dynamic loop unrolling, we're doing the similar thing as with the if statements - we create nested scopes (every next iteration of the loop potentially affects everything in the previous iterations), and since we can't know how many iterations are enough, we defer the decision to the C programmer writing original code. They should set the maximum unroll limit (which will not be less - condition will be evaluated exactly this number of times) using the `_unroll` variable. The variable can be set several times, and isekai always takes the most recent assignment for the number of unroll iterations.

Then the interesting transformation takes place - we're creating Conditional intermediate language node for every expression in the current scope - if the condition is true, we're taking the value from the current scope, if the value is false, we're taking the value from the parent scope - which is the equivalent as if the loop body never executed (for details, look into the comments in `unroll_dynamic_loop`).

4 Backend

The backend part of the compiler is coupled with the frontend through the intermediate representation which is produced by the frontend part. Based on the input arguments, arithmetic or boolean circuit generator is created and it's feed by the inputs and outputs, finally producing the text file with the circuit data (in text form).

4.1 Arithmetic Circuit

Formally, an arithmetic circuit is a DAG (directed acyclic graph), whose vertices are operators, each one having several (0..n) inputs and outputs, represented by the edges of the graph. They are encoded with .arith file format, coming from Pinocchio paper [PGHR13]. Example of running isekai to produce the arithmetic circuit would be:

```
isekai --arith=out.arith eqtest.c
```

That will produce the out.arith file with the following circuit:

```
total 14
input 0 # input
input 1 # input
input 2 # one-input
const-mul-0 in 1 <2> out 1 <3> # zero
const-mul-2 in 1 <0> out 1 <4> # multiply-by-constant 2
const-mul-5 in 1 <2> out 1 <5> # constant 5
add in 2 <1 5> out 1 <6> # #<Isekai::ArithAddReq:0x7fad84c91b00>
const-mul-neg-1 in 1 <4> out 1 <7> # zerop subtract negative
add in 2 <6 7> out 1 <8> # zerop diff
zerop in 1 <8> out 2 <10 9> # zerop #<Isekai::ArithAddBus:0x7fad84d03e10>
```

```
const-mul-neg-1 in 1 <9> out 1 <11> # zerop inverse
add in 2 <2 11> out 1 <12> # zerop result
mul in 2 <2 12> out 1 <13> # output-cast
output 13 #
```

In the context of arithmetic circuit, vertices and edges are called gates and wires respectively and we will use this naming from now on.

The main parts of the circuit are the first line that shows the total number of gates, and a line for each gate. A gate can be an operation of the form:

```
gate in input_wire_list out output_wire_list # comment
```

The table 1 below list the available gates.

Gate	Description	Remark
add	Addition gate	Outputs the addition of the inputs
mul	Multiplication gate	Outputs the multiplication of the inputs
const-mul-neg-xx	Minus multiplication times xx	xx is a constant, outputs the input times xx times -1
const-mul-xx	Multiplication times xx	xx is a constant, outputs the input times xx
zerop	Zero-Equality gate	Returns 1 if input not null, 0 else. The gate has an additional output wire for technical purpose.
split	Bit decomposition	Outputs the bits of the input

Table 1: Arithmetic gates

`wire_list` are the input/output of the command, expressed as a list of wires: number of wires, <wire indices>

Looking at `const-mul-neg-1 in 1 <4> out 1 <7>` we can see that this is multiply-by-minus-one gate which has one input and one output wire and they are 4 and 7. Looking in the next line we can see that wire 7 is the input for the Add gate and looking in previous lines, we can see that the input is fed from `multiply-by-constant 2` gate.

4.2 Bus

The entire procedure for generating the circuit from the graph representation is remarkably simple and it consists of several steps. The main component of the circuit is a gate, and a set of gates connected in a predefined manner is called a Bus. Bus has several properties, such as input wires count, output wires count and the gates that it contains. isekai feeds the intermediate representation into the Factory which produces a list of appropriate Bus Requests (one bus request for each node in the IR). Bus Requests is the request from the backend to lay down the most appropriate bus. As an example, `MultiplyBusReq` would yield a `MultiplyBus`, but it can also yield a `ConstantMultiplyBus` if isekai figures out that one of the inputs is constant. This way we keep Bus implementation minimal, without any cognitive overhead caused by the optimization.

Finally, everything is laid down and the factory will sort the buses based on the order of dependencies, with a bit of weight added so that inputs appear on the top and the output appears on the bottom, and isekai then just writes all buses - one gate per line.

Note that isekai supports both Arithmetic and Boolean circuits, but boolean circuits are much more restricted feature wise and since the code is much the same, we focus on the arithmetic circuits here.

Below are some examples, one can refer to the buses defined in isekai to see the full list (in the file `arithbuses.cr`).

Join bus
Convert a set of bits as input into the corresponding integer as output;

$$a = \sum_i a_i * 2^i \tag{1}$$

The ai are the input wires and a is the output. This bus generates the circuit for computing a using + and *

Constant Multiplication
Generates the appropriate const-mul-xx gate

Bit operations
Performs bit operations using arithmetic circuit:
a AND b: a*b
NOT b: 1-b
a OR b: 1-(1-a)(1-b)
a XOR b: a+b-2a*b
a NAND b: 1-a*b
Left,right shifts

Bus requests
CmpLTReq: a¡b is done by 'split'-ing (a-b) into bits and returning its sign bit

... there are many more bus in busreq.cr and arithbusreq.cr files

4.3 Trace, FieldOp and Wire

Trace is an entity that connects two buses together and it has a single property: a trace type. Trace type might be arithmetic or boolean, and its use is to tell isekai what kind of gates we may apply. For example, if we look into the `ConditionalReq`, we'll see that it will generate three buses - the bus for the conditional value, which is a boolean type, and one arithmetic bus for both true and false. The arithmetic bus types are here chosen as their implementation is much easier for this operation, then for the boolean one. Of course, we also need to connect buses of a different type, and for that we have two special buses - `JoinBus` and `SplitBus` which convert from one bus to another.

The primitive for connecting everything together is Wire. Wire is rarely used alone, it's mostly referenced in a `WireList` object, which holds the array of wires. For example, see ArithmeticConditionalBus's `get_field_ops` method's definition:

```
trueterm = @wire_list[0]
minuscond = @wire_list[1]
negcond = @wire_list[2]
falseterm = @wire_list[3]
result = @wire_list[4]
return [
    FieldMul.new("cond trueterm",
        WireList.new([@buscond.get_trace(0), @bustrue.get_trace(0)]),
        WireList.new([trueterm])),
    FieldConstMul.new("cond minuscond",
        -1,
        @buscond.get_trace(0),
        minuscond),
    FieldAdd.new("cond negcond",
        WireList.new([@board.one_wire(), minuscond]),
        WireList.new([negcond])),
    FieldMul.new("cond falseterm",
        WireList.new([negcond, @busfalse.get_trace(0)]),
        WireList.new([falseterm])),
    FieldAdd.new("cond result",
        WireList.new([trueterm, falseterm]),
        WireList.new([result]))
    ]
```

This method allocates five wires internally one for the bus and it gives each wire a name. Then it proceeds to create the series of `FieldOps` - which are the gates/operations in the fields - and it connects them with the wires. For example, look how the input trace is multiplied by -1 and then the resulting wire - minuscond is connected to the Adder which adds the (-1 * cond) with 1 and its result is then plugged in to the multiplier which is then connected to the adder and the final result is in the fourth wire.

4.4 Board

Everything we described so far happens on a hypothetical board. The board, as in the real life, describes the constraints our circuit is working with and holds the busses. When board is allocated, we always place a 1 bus which provides constant value of 1, from which all other constants are calculated feeding 1 into constant multipliers. Board also holds optional 0, but the most important property of the board is the bit width. This tells us how many wires we can fit in a bus and by that we can tell if there's a possibility to overflow.

4.4.1 Bit Width

It is very important to track the bit width of every wire. This property is defined at the bus level and tells how many wires can fit in the bus. In isekai, a bus tells the maximum number of active bits based on the input using the `get_active_bits()` function. For instance, multiplying two 32 bits can produce a 64 bits value, adding two 32 bits can

produce 33 bits value, etc... Note that the wire size has a maximum value (typically 128 or 256 bits).
For example, this is the definition of `ArithAddBus::active_bits`:

```
max_bits = Math.max(bus_left.get_active_bits(), bus_right.get_active_bits())
@active_bits = max_bits + 1
```

And here's the same for the multiplying gate:

```
@active_bits = bus_left.get_active_bits() + bus_right.get_active_bits()
```

So, based on the operation, we can predict the maximum number of significant bits and see if the operation will overflow.

The circuit have a maximum bit width and there is a setting (`ignore_overflow`) to check when there is potential overflow. In that case, the wire is 'split' into bits and we keep only the first 32 bits. Based on the bit width, we define other useful values, which are stored in `bitwidth.cr` file, such as sign bit position, negative one constant, etc.

4.5 Inputs

The inputs of the circuit must correspond to Input and NzikInput arguments of the input C function. Then there is be an additional wire which correspond to the constant 1. There can also be some additional internal variable depending on the gates. All wires must be only integers. For instance the following Input:

```
Struct Input {int my_values[5]; int my_var;}
```

should be translated as 7 wires; 5 for the `my_values`, one for and then the wire for the constant 1. If the structure has float, you will get an error. If the function has NzikInput, their corresponding wires should come after the constant 1. Circuit inputs are expressed as `'type of wire', 'wire indice'` where type of wire can be:

- input (Input or constant 1)
- nzikinput (from NzikInput)
- output

isekai generates the input file that corresponds to the circuit (at the same time the circuit is generated). It is simply a list of values for each (nzik)input wire (one value per line). isekai generates this file with the .in suffix (from the arithmetic circuit filename). Note that the inputs of the C program are not the same than the inputs of the circuit because some gates may create new internal inputs, and there is at least the constant 1 wire that must have 1 as value in the input file. The end-user can specify the inputs of the C program by providing a file (with .in suffix to the C program filename). If it is not provided, isekai assumes all the values are zero. isekai will add some comments using the # character. The inputs corresponding to the Input structures have # input as comment, and the constant 1 has one-constant as comment.

4.6 Wire Ordering

The wires will be written by isekai in the following order: the inputs have the highest ordering of all, then normal wires and then the output(s). This means that the circuit will start with the 'Inputs'. For example, if you have a struct with 8 fields in the input, they are all going to be there first, and then the wire for the one-input. In the code that the ordering if done with the tuple `‘(major_order, order)‘`, where order is assigned by the board - it's just a sequence, and 'major' is set by the user at the construction point. (cf. function `orders` in the `Bus` class) In `‘bus.cr‘` you have:

```
class Constants
     @@MAJOR_INPUT = 0
     @@MAJOR_INPUT_ONE = 1
     @@MAJOR_INPUT_NIZK = 2
     @@MAJOR_LOGIC = 3
     @@MAJOR_OUTPUT = 4

     @@ARITHMETIC_TRACE = "A"
     @@BOOLEAN_TRACE = "B"
 end
```

This means that first we have all the input buses, then then special `nizk_input` (if present), and then all logic buses and finally the output bus.

4.7 Bus request and factory

For every node in the intermediate representation, the Factory creates an appropriate Bus request, as in the following example in `ArithFactory`:

```
def make_req(expr, type : String) : BaseReq
    case expr
    when .is_a? Input
         result = ArithmeticInputReq.new(self, expr.as(Input), type)
     when .is_a? NIZKInput
         result = ArithmeticNIZKInputReq.new(self, expr.as(NIZKInput), type)
     when .is_a? Conditional
         result = ArithConditionalReq.new(self, expr.as(Conditional), type)
     when .is_a? CmpLT
         result = CmpLTReq.new(self, expr.as(CmpLT), type)
     when .is_a? CmpLEQ
         result = CmpLEQReq.new(self, expr.as(CmpLEQ), type)
    when .is_a? CmpEQ
        result = CmpEQReqArith.new(self, expr.as(CmpEQ), type)
     when .is_a? Constant
         result = ConstantReq.new(self, expr.as(Constant), type)
     when .is_a? Add
         result = ArithAddReq.new(self, expr.as(Add), type)
```

The main reason for having the requests is that to tell isekai that we're requesting a bus to be added. However, a bus may be dependent on other buses, and every Bus request specifies these dependencies. Let's see `ArithConditionalBusReq` for example - where the bus depends on a bus layered down for condition and for two outcomes:

```
private def reqcond()
   return LogicalCastReq.new(@reqfactory, @expr.as(Conditional).@cond, Constants::BOO
end

private def reqtrue()
   return @reqfactory.make_req(@expr.as(Conditional).@valtrue, Constants::ARITHMETIC_
end

private def reqfalse()
   return @reqfactory.make_req(@expr.as(Conditional).@valfalse, Constants::ARITHMETIC
end

def natural_dependencies()
   return [ reqcond(), reqtrue(), reqfalse() ]
end
```

Here we also see that we're recursively create more bus requests, for every branch in the intermediate language conditional node.

The Factory object is also a `collapser` - the class that we already used in the frontend. As we said, for each node, Collapser first calculates and resolves all dependencies before outputing the object itself, and it's object's responsibility to specify its dependencies. This is where bus request fits in - bus request specifies all dependencies that are needed for the given bus. For example, we may have `ArithAddRequest` which will lay down `ArithAddBus`, but this bus also depends on left and right operand's buses. isekai starts from the output, and using Collapser (contained in the Factory itself) recursively resolves all the buses needed for all the dependencies.

4.8 Backend Flow

Once we are familiar with all the components, it's very easy to inspect the flow of the backend. It can be separated in a series of preparation steps, with the last large recursive step:

1. An object of `ArithFactory` is created, which creates its superclass object - `ReqFactory`
2. `ReqFactory`'s constructor will setup the board, lay down 1-constant bus and 0 bus
3. Starting from the output, `OutputBusReq` will be issued - which will then be collapsed, and all dependencies to the input will be recursively resolved and for each the appropriate bus request will be issued.
4. The mapping between the bus request and intermediate language can be found in a switch in `ArithFactory` (and other factories), in the method `make_req`

5. We'll add all the inputs on the board, even if they are not used, that is - even if the output doesn't depend on them, so they were not recursively resolved in step 3.
6. We'll sort the buses in the opposite order they were requested, starting with the input buses and ending with the output bus
7. We'll number each wire from/to every bus
8. And finally, we'll print out every `FieldOp` object with all the wiring information into the output file.

4.9 R1CS reduction

Rank-1 Constraint Systems are bi-linear equations of the following form

[BSCG+13]:

$$(a_1x_1 + a_2x_2 + ... + a_nx_n) * (b_1x_1 + b_2x_2 + ... + b_nx_n) = (c_1x_1 + c_2x_2 + ... + c_nx_n) \quad (2)$$

They are derived from the arithmetic circuit using simple rules:

- Every wire of the circuit correspond to one variable x_i.
- Every gate of the circuit is translated to one (or more) bi-linear equation in order to represent the relation between the output and input wires.

For instance an addition gate will give a simple equation resulting from the addition:

```
add in 2 <2 11> out 1 <12>
```

is translated into

$$x_{12} = x_2 + x_{11} \quad (3)$$

That way, the whole circuit can be completely represented with such equation system, where the solution to the equations are the values taken by the wires of the circuit during the program execution. In order to be able to apply cryptography operations over these equations, we restrict them to be inside a field $\mathbb{Z}/p\mathbb{Z}$ where p is a big prime number (usually 256 bits), chosen before doing the R1CS reduction. The values taken by any wire must be less than the prime number, this is why we have a maximum bit width for the wires. Currently the prime number is fixed in isekai but we should be able to choose it because it is a parameter for the proof system we want to use in the next step.

The reduction from circuit into R1CS is done with `libsnark` [lib], using the `get_constraint_system_from_gadgetlib2()` function. We have implemented a C++ library providing a C wrapper over libsnark for isekai. libsnark is taking an .arith file in argument and libsnarc will save the generated R1CS in j1cs format [Dre19], which is an open json format for R1CS that we have defined for interoperability. Inputs (public and private), outputs and witness are also generated by libsnark during this phase, and libsnarc writes them in a json file which have the '.in' extension in addition to the R1CS file name.

4.10 Prover

isekai aims to handle most popular zero-knowledge proof systems, by interfacing them using the R1CS file produced in the previous step. Right now we have interfaced with libsnark in order to generate zk-snark proofs. We refer to the libsnark documentation [lib] for more information.

isekai can also be used for verifying a proof and generating the trusted setup.

4.11 Next steps

There are many points on which we would like to work; adding new languages, support more features of a language and adding more zero-knowledge libraries.

4.11.1 Programming languages

Although it should not be too difficult to support additional programming languages since most of the code can be re-used once we have the AST from the input program, generating the AST is a job already done by usual compilers. Then it would be much more beneficial for us to look at compilers supporting already several languages. In particular we have started to look at LLVM (http://llvm.org/), a well-known compiler, and plan to support LLVM IR so that we will be able to support automatically all languages supported by LLVM, in theory. In practice it will not be so easy because we will not support all of the LLVM instructions anyway.

The most challenging part of supporting LLVM IR is due to the arbitrary control flow graph (CFG). Compiling a CFG into a structured programming language with conditional statements, switch-like statement and loops with multi-level break statements, but no goto-like statements (such as JavaScript) efficiently is already hard, but in our case, we not only don't support goto statements, but also break, continue and return statements.

Basically, we would like to come up with an algorithm that can convert LLVM IR code from C99 source code without goto, break, continue or return statements into if/while statements and implement it in isekai along with a LLVM parser. Then we would use LLVM to support additional languages by compiling them in isekai into LLVM bytecode.

Another popular compiler that target a goal similar to LLVM by compiling programming languages into a universal bytecode which is mostly hardware independent, is Web Assembly (https://webassembly.org/).

Web Assembly is a new technology which is bringing native speed and multiple languages support to the web. Therefore we would like to have an isekai frontend which can read and interpret WebAssembly files (wasm), so that we can use the isekai backend to generate an arithmetic representation of the WebAssembly code.

4.11.2 High level language features

There are many limitations regarding the features from the C language that you can use in the input program for isekai. For instance, we do not support function call, arrays

nor floating-point types. As explained in the previous paragraph, we also do not support goto, break and return statements so we would also like to investigate on ways to support arbitrary control graph of LLVM.

One should notice that in order to support additional features, we will need in the end to generate R1CS for these features. But these are not supported by the library that we use for generating the R1CS. So the prerequisite here would be first to implement our own implementation of the R1CS generation and to be able to combine R1CS files together. We would also like to have some R1CS tester in order to validate our R1CS generation by comparing the outputs from R1CS and the one from the program execution, coming from random inputs. The outputs could be retrieved from the R1CS generation or (if practical) by solving the R1CS.

One very useful feature that is also no supported is memory usage. We would like to support first immutable memory allocation, and then mutable memory by creating new variables at each modification. Finally, we would like to generate tinyRAM circuit [BSCG$^+$13] as well as implement the tinyRAM approach in the context of LLVM IR.

We strongly believe that our goal of having a tool usable by non zk-specialist is dependent on supporting these features.

4.11.3 Zero-Knowledge Libraries

We would like to support popular zero-knowledge implementation and have already started to look at Bulletproofs [Dal]. For this we would need to generate R1CS with a prime number used by the Bulletproof implementation, either by defining a custom curve in libsnark or by implementing our own generation (cf above), then we will need to parse the generated R1CS json file in rust, call the Dalek API with this and finally create a C wrapper over the rust implementation to be called by isekai.

The other implementations that we will target next are ZK-STARKs [Sta], but we need to wait for the complete API to be available, and AURORA, which are using structures similar to libsnark.

Finally, we would also like to investigate homomorphic encryption and see how such techniques could be integrated within isekai.

4.11.4 summary

We summarize below the points that we would like to investigate.

- LLVM IR
  - LLVM IR to if/while statement algorithm
  - LLVM Bytecode parsing
  - Additional languages to LLVM Bytecode
- Web Assembly
  - Wasm parsing

  - Wasm to arithmetic circuit
- R1Cs Composer
  - Combine R1Cs files
  - Generate custom R1CS for special gates
- R1Cs Tester
  - Generator
  - Solver
- Programming language features
  - Float
  - Arrays
  - Function call
  - Control Flow
  - Memory
    * Read-only allocation
    * Read-Write
    * TinyRAM
    * TinyRAM for LLVM Bytecode
- Bulletproof
  - Custom curve for libsnark
  - J1cs in rust
  - C-Wrapper for Dalek
- ZK-STARKs
- Aurora
- Fully homomorphic encryption
  - State-of-the-art
  - Existing tools
  - isekai integration

References

[BSCG+13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In *CRYPTO (2)*, volume 8043 of *Lecture Notes in Computer Science*, pages 90–108. Springer, 2013.

[Dal] Dalekś bulletproof. https://github.com/dalek-cryptography/bulletproofs/.

[Dre19] Guillaume Drevon. J-r1cs : a json lines format for r1cs. http://www.sikoba.com/docs/SKOR_GD_R1CS_Format.pdf, 2019.

[IB19] Simon St. Laurent Ivo Balbaert. *Programming Crystal*. The pragmatic Programmers, 2019.

[lib] libsnark. https://github.com/SCIPR-Lab/libsnark.

[Man19] Manas. The crystal programming language. https://crystallang.org/, 2019.

[PGHR13] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. *IACR Cryptology ePrint Archive*, 2013:279, 2013.

[Sta] Zk-starks. https://github.com/elibensasson/libSTARK.

[Tea19] Clang Team. Introduction to the clang ast. http://clang.llvm.org/docs/IntroductionToTheClangAST.html, 2007-2019.